

Computer Science Department

TECHNICAL REPORT

AN ERROR-BASED TESTING STRATEGY

by

ELAINE J. WEYUKER

JANUARY 1981

REPORT NO. 027

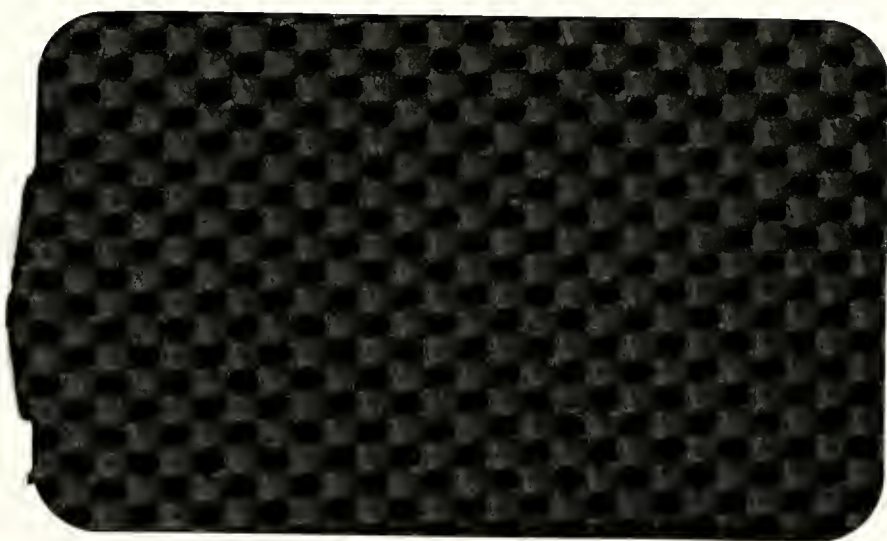
NEW YORK UNIVERSITY



Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

CSD TR-027

C. Z



AN ERROR-BASED TESTING STRATEGY

by

ELAINE J. WEYUKER

JANUARY 1981

REPORT NO. 027

1000 0000

1000 0000

ABSTRACT

An error-based testing strategy produces test data whose purpose is to expose the presence of errors likely to occur in the program. To design such tests, a two-phase strategy is developed which uses all relevant sources of information, including the program's specifications, the algorithm, the problem's input and output data structures, the program's code, and knowledge of common programming errors. Three primary areas are considered: the determination of likely errors, the design of tests to detect these errors, and the demonstration that these tests do in fact expose the presence of the specified errors.

1. INTRODUCTION

Testing is the most common way of gaining confidence in the correctness of software. Despite a long history of practical testing experience, it is only during the last five years that researchers have attempted to formulate a theoretical foundation for testing.

The initial steps in this direction were taken by Goodenough and Gerhart [5], who formulated an ambitious theory which described conditions under which a program can be determined to be correct by testing. They defined a set of inputs T to be an ideal test for a program P , relative to specifications S , if the correct performance of P on T implies the program is correct on its entire input domain. A test selection criterion is a rule used to select sets of tests.

Their fundamental theorem of testing gives sufficient conditions for a criterion to select ideal tests. The theorem's conditions can be used as guidelines for developing test cases, but the conditions are so strong that they are essentially impossible to fulfill in practice. Stimulated by this work, subsequent research aimed at finding results of more limited scope, but with broader applicability.

Three types of problems must be dealt with when defining a strategy based on the concept of ideal tests. They are: formal unsolvability, impracticality, and unrealistic assumptions. As we shall see, theories with less ambitious goals as well as various methodologies used in practice, must also face these problems.

A typical unsolvability problem for the theory of ideal tests is described in [10]: There is no computable procedure which, given an arbitrary program P and specifications S with domain D , can produce a proper subset T of D which is an ideal test for P relative to S and D .

An example of the second type of problem is pointed out by Hamlet [7], Howden [10], and Weyuker and Ostrand [20], who mention problems related to the "ideal empty test." If a program is correct, then any test, including the empty set, is ideal! The problem of testing arbitrary implementations of a given specification is also discussed in [20], where it is shown that no test is ideal for a specification unless it consists of the entire input space.

An instance of the unrealistic assumption problem is that practically all work on testing, both theoretical and applied, has assumed the existence of an oracle capable of determining the correctness of outputs produced from test inputs. As discussed in [19], however, this is not a realistic assumption in many situations. This type of problem calls for greater familiarity with the actual situations which arise during software testing.

2. APPROXIMATIONS TO IDEAL TESTS

As the difficulties with a testing theory which attempts to show correctness of unrestricted programs have been recognized, various approximations to the ideal test concept have been formulated. These approximations basically fall into two categories. The first way of proceeding is to place restrictions

on the form of programs to be considered in the hope that an "interesting" class of programs can be found for which the typical types of questions (equivalence, halting, reachability) are decidable in reasonable amounts of time. By an "interesting" class of programs we mean one which models the programs that people actually write. It is our experience [16, 17, 18] that the types of restrictions one must place on programs in order to get solvable problems are so severe that one is left with trivial or at best unnatural classes of programs. Examples of this way of proceeding include Howden's algebraic program testing [11] and White and Cohen's domain testing strategy [21].

Given a program P in a restricted class of programs $\mathcal{P}$, Howden [11] describes characteristics of a test set which can be used to show that P is correct. Programs in $\mathcal{P}$ have restricted types of loop constructs, contain no branch statements, and there is no interaction between index variables and computational variables. If $\mathcal{P}$ contains some correct program Q , and if Q 's computation and the computation of P iterate loops in a certain way, then the correctness of P on an appropriate set of test data implies the correctness of P on its entire domain.

The "domain testing strategy" developed by White and Cohen [21] aims at detecting domain errors. An input space domain is the set of program inputs which satisfy a path condition. The entire input space of a program P is partitioned into input domains; a domain error exists if P can be transformed into a correct program by a sequence of changes at least one of which modifies a domain boundary. White and Cohen study classes of

programs whose domain boundaries are produced by simple (a single relational expression), linear predicates. Assuming that coincidental correctness does not occur, and that there are no completely omitted input domains, White and Cohen show that finitely many inputs suffice to test for all domain errors. Tai showed that this result applies to the class of programs containing linear assignment statements and alternation statements with linear, simple predicates [14].

The second type of approximation to an ideal theory is to accept that one is testing only for the presence of a specified subset of all errors. By carefully selecting this class of errors, one is able to feel confident about the likelihood of the program's correctness, even though this has not actually been guaranteed.

Weyuker and Ostrand [20] introduce the concept of a revealing subdomain, a subset of the program's input space in which either every element is processed correctly, or every element is processed incorrectly. If the input space can be partitioned into revealing subdomains, then an ideal test simply consists of a single element from each subdomain. In order to use this notion to test programs practically, one attempts to find subdomains which are revealing for specific errors. Correct execution of an element from such a subdomain guarantees the absence of the specified error on that subdomain.

Another testing strategy which has been proposed is known as the program mutation method [1, 3]. This system makes a series of minor changes to the program being tested, creating a set of

programs known as mutations. Some of these modifications cause program errors, while others simply yield equivalent programs. A proposed set of test data is considered adequate if it causes every inequivalent mutation to give an incorrect answer on some input in the set. What the authors have done is to implicitly define what they consider to be the class of most likely simple errors. By showing that these errors do not occur, they have not guaranteed the absence of all errors, but rather that the program is either correct or radically incorrect. Since the authors assume that the program being tested was written by a "competent programmer", i.e. a person who writes programs which are "close" to being correct, the second alternative can be eliminated.

Several problems arise when trying to use the program mutation method as the basis of a pragmatic testing strategy. The first is the number of alternative programs which must be distinguished from the original. The authors have stated that for an n statement program, the system generates on the order of n^2 mutations [1]. Obviously there will be a very large number of programs to be run and outputs compared for even a moderate sized program. Also, some of the mutations will be equivalent to the original. But how are these equivalent programs to be detected? Program equivalence is in general a recursively unsolvable problem, and although there are many cases for which it is possible to determine that two given programs are equivalent, it is frequently difficult to prove this.

In addition, there is the question of whether or not what the authors have called the "coupling effect" and "competent

programmer" assumptions really hold. They claim that "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors," and that a program written by a competent programmer "differs from a correct program in 'simple' ways."

Other work based on testing for specific types of errors includes the recent papers by Foster [4] and Howden [13].

Most proposed testing strategies share another problem; they rely entirely on the program's code to develop test data. Howden is one of the few authors who advocate the use of program independent information; his functional testing approach [12] is in fact largely program independent. We agree with Howden's conclusion that "functional and structural testing should be viewed as complementary rather than competing techniques." Surely the use of other sources of information such as the specifications, knowledge of the algorithm and data structures used, and common error statistics, can only enhance the error detecting capabilities of a testing strategy. In particular, using solely the program code, it is unlikely that missing sections would be detected. This is an important problem since omitted code has been determined to be a very common source of errors. The TRW study on software reliability [15] states: "Logical errors, the major category with the most occurrences for all projects, fell predominantly in the missing logic or condition test detailed category for all projects. Percentages ranged from 39.6 percent ...to 76.6 percent.... Of all the

errors encountered this one represents the most significant found in the whole study".

3. THE TESTING STRATEGY

In general, the goal of testing should be to expose errors, not to guarantee correctness. A testing theory and associated methodology should not require that the user know which errors actually occur in the program. It is reasonable, however, to assume that the user does have some information and experience concerning which errors are most likely to occur. An error-based strategy is one in which an attempt is made to derive test data to determine the presence or absence of specified errors. Such a strategy is likely to be fruitful provided we can judiciously select the errors for which we will test. This may be done by examining the program's specifications, the algorithm, the structure of the program, and by drawing on programming experience. We note that although hardware testing has traditionally been error-based, until recently software testing has not generally proceeded in this manner.

In a recent paper discussing Ada compiler validation [6], Goodenough states "a set of tests aimed at detecting particular errors of omission and commission are more likely to actually detect errors than tests designed without any particular error in mind." The recognition of this by people involved in the validation of large pieces of software underscores the importance of developing good, usable error-based testing strategies.

We now outline such a strategy. In particular, the strategy

consists of defining the class of most likely errors, and then creating test data to determine whether or not each of the selected errors is present. What is being done in effect is to distinguish the program as written from those programs which would be most likely to occur if the given program were not correct. In the next section we demonstrate the details of the strategy with an example.

In order to identify likely errors in a program, we employ a two phase strategy. The first phase relies on information other than the program. During this phase, we attempt to identify program independent boundary conditions for the input domain, as well as other input special cases which are particularly error prone. Since the potential errors identified during this phase are program independent, they can be used to test any program which alleges to fulfill the given set of specifications. Furthermore, phase one tests are useful throughout the testing/debugging period. This is in contrast to Goodenough and Gerhart's notion of an ideal test [5] which is totally program dependent. A test set which is ideal for a given program may not remain so once an error has been located and removed. This was discussed and illustrated in [20]. Thus, each time a program is modified during debugging, new test criteria and tests must be developed if using the Goodenough and Gerhart theory. This is unnecessary for our phase one test data. As was noted above, this problem of reliance upon program dependent test data is certainly not unique to the Goodenough and Gerhart strategy.

The second phase of the strategy relies mainly on the

program itself, and a knowledge of common programming errors. We systematically consider program statements and, based on past experience, try to identify common errors. For example, a frequent error in a FOR loop which ranges from M to N is that one of the indices is off by one.

We next try to determine what elements of the domain would have been affected and what the effect would have been if such an error were present. Thus if the FOR loop ranges from M to N but should have ranged from M to N+1, the effect might be that one too few elements of an array are processed, and test data should be selected which will reflect this insufficient processing. In the next section we demonstrate our strategy with an example.

4. EXAMPLE

We now apply the strategy outlined in Section 3 to a simple program, and create a set of errors which are likely to occur. Since all the errors have the form:

Whenever condition, then the program fails,
it is simple to define a test selection criterion for each.

SPECIFICATIONS: Locate all instances of the minimum value in an N-location array A, which has been previously initialized.

ALGORITHM: Locate the first instance of the minimum value in the array, and then compare to it each element with higher array index to determine other locations of the minimum.

PROGRAM:

```

1  MIN:=A[1]
2  IND:=1
3  FOR I = 2 UNTIL N DO
4      IF A[I] < MIN THEN
5          MIN:=A[I]
6          IND:=I
7      ENDIF
8  ENDFOR
9  PRINT(MIN,IND)
10 FOR I = IND+1 UNTIL N DO
11     IF A[I] = MIN THEN PRINT I ENDIF
12 ENDFOR

```

Phase 1 Program Independent Potential Errors and Tests

- a) The program doesn't find the minimum if it occurs in the first location of the array - test on an array with the minimum in A[1].
- b) The program doesn't find the minimum if it occurs in the last location of the array - test on an array with the minimum in A[N]
- c) The program doesn't find all occurrences of the minimum - test on an array with more than one occurrence of the minimum.
- d) The program doesn't find the minimum (it finds the second smallest or maximum, for example) - test on an array with more than one distinct value.
- e) The program doesn't work on an array of size one - test on an array of size one.

Phase 2 Program Dependent Potential Errors

line 3: FOR loop has the wrong lower bound

f) should be FOR I = 1 UNTIL N DO

FOR loop has the wrong upper bound

g) should be FOR I = 2 UNTIL N-1 DO

h) should be FOR I = 2 UNTIL N+1 DO

line 4: wrong predicate

i) should be IF A[I] < MIN THEN

j) should be IF A[I] > MIN THEN

k) should be IF A[I] ≥ MIN THEN

l) should be IF A[I] = MIN THEN

line 10: FOR loop has the wrong lower-bound

m) should be FOR I = IND-1 UNTIL N DO

n) should be FOR I = IND UNTIL N DO

FOR loop has the wrong upper bound

o) should be FOR I = IND+1 UNTIL N-1 DO

p) should be FOR I = IND+1 UNTIL N+1 DO

line 11: wrong predicate

q) should be IF A[I] < MIN THEN

r) should be IF A[I] < MIN THEN

Possible effects of the above errors and associated tests.

f) Missing the minimum if it is in A[1] - test on an array with the minimum in A[1].

g,o) Array index out of bounds - test on any array.

h) Missing the minimum if it is in A[N] - test on an array with the minimum in A[N].

- i) Not finding the correct instance of the minimum if more than one occurrence - test on an array containing more than one occurrence of the minimum.
- j,k,l) Not finding the minimum element - test on an array with more than one distinct value.
- m) If the first occurrence of the minimum is in $A[k]$, and $A[k+1]$ or $A[k+2]$ also contains this value, the program misses them - test on an array with the first two occurrences of the minimum within three locations.
- n) If the first occurrence of the minimum is in $A[k]$, the next occurrence in $A[k+1]$, the program misses $A[k+1]$ - test on an array with the first two occurrences of the minimum in successive locations.
- p) Missing the minimum if there is more than one occurrence of it, including an occurrence in $A[N]$ - test on an array containing at least two instances of the minimum including an instance in $A[N]$.
- q,r) There should never be a value at this statement which is less than MIN. If there is, then MIN is not found correctly - test on an array with the minimum not in the first location.

The two-phase strategy thus leads to the following set of likely errors for the program:

- E1: If the minimum appears in the first location of the array, then the program does not return the correct answer.
- E2: If the minimum appears in the last location of the array, then the program does not return the correct answer.

E3: If the array contains more than one instance of the minimum, then the program does not find all the instances.

E4: If the array contains more than one distinct value, then the program does not find the minimum.

E5: If the minimum is not in the first element of the array, then the program does not find the minimum.

E6: If the array contains at least two copies of the minimum, and the first two are in consecutive locations, then the program does not print all the locations of the minimum.

E7: If the array contains only one element, then the program does not return the correct answer.

Defining the test data criteria is now a straightforward procedure:

C1: Test contains an array with the minimum in the first location.

C2: Test contains an array with the minimum in the last location.

C3: Test contains an array with more than one occurrence of the minimum.

C4: Test contains an array with more than one distinct value.

C5: Test contains an array with the minimum not located at the first element of the array.

C6: Test contains an array with at least two instances of the minimum value; the first two occurrences in consecutive locations.

C7: Test contains an array of size one.

We have now only to show that if the program contains error E_i , and if T is a test which satisfies criterion C_i , then the program will not run successfully on T . But by the definition of the E_i 's and C_i 's, this is immediate.

Now let T be a set of tests such that each of the criteria C_i is satisfied by at least one test in T . If the program runs successfully on T , then we may conclude that it does not contain any of the specified errors E_i .

A set of three arrays is sufficient to meet all the criteria, and determine the presence or absence of the specified errors. The table of Figure 1 shows the seven criteria satisfied by three arrays which would form a test set.

Note that successful execution of the program on these three test cases does not prove the program's correctness on its entire domain, but does show that it does not contain the seven specified errors. Since the specified errors were selected precisely because they are the ones we considered most likely to occur, and this determination was made using a complete range of available information, we conclude that the program is likely to be correct. The degree of likelihood is of course a function of the way the likely errors were selected.

There are several important distinctions between our strategy and the mutation method. Philosophically, the mutation system was intended as an adequacy measure, not a test data selection strategy. Furthermore, the mutation method is

	C1	C2	C3	C4	C5	C6	C7
ARRAY1 = (3 1 2)				X	X		
ARRAY2 = (1 1)	X	X	X			X	
ARRAY3 = (1)	X	X					X

Figure 1

completely program dependent. As mentioned above, this is a weakness shared by almost all other proposed adequacy measures and testing methodologies. Clearly the inclusion of many information sources can only help to locate errors. On the positive side, however, it is precisely this complete reliance on program dependent information which permits the set of mutated programs to be well defined and mechanically generable, obviously desirable features.

Another difference between our strategy and the mutation method is that no guidance is given by the mutation system as to how to select the desired test data. In contrast, our strategy operates with an explicit eye towards test data generation. Rather than modifying the program and attempting to locate test data which is processed correctly by the original program, but incorrectly by the modified program, we ask: if our program were incorrect in a specified way, what data would be processed incorrectly?

Admittedly our process is much more ad hoc than the mutation method. We feel, however, that the program independent portion of the testing process is inherently ad hoc, and any completely mechanizable methodology is incapable of utilizing all available information sources, and is therefore not likely to be as effective as one which does. Indeed, the authors of the mutation method state [3]: "there is certainly no need to apologize for applying ad hoc strategies in program testing. A programmer who considers his problems well and skillfully applies appropriate techniques to their solution - regardless of where the techniques

arise - will succeed."

We conclude this section by considering a minor modification to the above program which renders it incorrect. We then apply both path testing and our error-based testing strategies and compare their effectiveness. Assume the program is as written above, except that statement 4 is replaced by:

IF A[I] < MIN THEN

Since this is one of the potential errors which we explicitly identified as likely (i) during phase two, our strategy will obviously detect it. Furthermore, since the effect of this error is to cause the program to ignore multiple occurrences of the minimum array value, it will also be detected by one of the phase one program independent tests (c).

What types of tests are generated when a path testing strategy is applied to the modified program? Strictly speaking, path testing requires that every executable path through a program be traversed at least once. Execution sequences which differ only by the number of times a loop is iterated are considered distinct paths. Thus, many programs with loops will contain a potentially infinite number of paths. A testing strategy which requires each distinct path to be traversed is obviously not realistic for such programs, and several more practical approaches have been proposed. One commonly used restriction is the placement of a (usually small) upper bound on the number of times a loop must be iterated [2, 9]. A second

approach is the boundary interior method [8]. With this technique, two paths are considered equivalent (for the purpose of path testing) if they are identical outside each loop, and if, during the first traversal of each loop body, they follow the same subpath.

Using the latter of these approaches, the test data {1, 12, 21, 213, 231} traverses one path from each executable path group, as demonstrated in Figure 2, where loop 1 is the loop of statements 3-8, and loop 2 is the loop 10-12.

Lines 2, 3, 6, and 8 of the chart of Figure 2 represent infeasible paths for the (modified) program. If the incorrect program is tested on the above set of inputs, the output is correct for every array in the set. Thus this set, which fulfills the boundary interior criterion, does not expose the error. Note also that a subset of this data would fulfill the loop criterion discussed above with an upper bound of 1. Similar data will suffice for larger upper bounds.

The reason that these path testing strategies do not expose this error is that these structure-based testing strategies do not take into account the fact that once the input domain has been partitioned by a set of path conditions, there are values within a subdomain (particularly those at the boundaries) which are significantly more error prone than other data in the subdomain. Since these strategies do not mandate (or even encourage) the testing of such data, it is likely that many errors may be missed if one relies on such strategies. Since path testing subsumes branch and statement testing, the two most

Number of times loop 1 is traversed	Result of Interior test (line 4) on first iteration	Number of times loop 2 is traversed	Result of interior test (line 11) on first iteration	Test Data
0		0		1
0		≥ 1	yes	-
0		≥ 1	no	-
≥ 1	yes	0		21
≥ 1	no	0		231
≥ 1	yes	≥ 1	yes	-
≥ 1	yes	≥ 1	no	213
≥ 1	no	≥ 1	yes	-
≥ 1	no	≥ 1	no	12

FIGURE 2

commonly used structure-based testing strategies, we are encouraged in our conviction that error-based testing is more likely to expose errors than other popular program testing techniques.

5. CONCLUSIONS

Ideally, a testing strategy would define test data which expose errors whenever the program being tested is incorrect. Realistically, however, we cannot expect to be able to prove programs correct by testing. We have described a testing strategy which has two important characteristics. The first is that it approximates the ideal goal by testing for the presence or absence of errors specified by the tester. The second important point is that our strategy relies on both program dependent and program independent information. This is in contrast to many strategies which rely on only one of these types of information.

Our testing strategy aims at devising tests by considering likely potential errors. We described a two-phase strategy for identifying errors and developing error-based tests. The first phase attempts to identify potential program independent errors, while the second phase examines the given program's code for likely errors. Each identified error is given a uniform description, from which a test criterion to expose the error is defined. Test data are then selected by choosing a minimal set of inputs which satisfy all the criteria.

An important benefit of our strategy is that many potential

errors can be defined in a program independent way, and the tests can be applied to any attempted program solutions of the program. When these tests are applied to modified or corrected versions of the original program, they will expose the same errors. Their successful execution still implies that the new version is free of the specified types of errors. Program independent tests are also more likely to expose certain errors, such as missing sections of code, than structure-based tests.

Much work still needs to be done. The greatest need is for studies to determine the most common types of errors. This information would greatly enhance the value of our testing strategy and others such as those discussed in Section 2. Nevertheless, we believe our testing strategy provides a reasonable perspective and can be used now by relying on a program tester's experience to identify likely errors.

REFERENCES

- [1] A.T. Acree, R.A. DeMillo, T.J. Budd, R.J. Lipton, and F.G. Sayward, "Mutation Analysis," Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Sept 1979.
- [2] R.S. Boyer, B.E. Elspas, and K.N. Levitt, "SELECT--A Formal System for Testing and Debugging," Proc. of the Int. Conf. on Reliable Software, Los Angeles, April 1975, pp. 234-245.
- [3] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, 11(4), April 1978, pp. 34-41.
- [4] K.A. Foster, "Error-Sensitive Test Cases Analysis," IEEE Trans. Software Eng., Vol.SE-6, May 1980, pp.258-264.
- [5] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Testing: Data Selection Criteria," in Current Trends in Programming Methodology, Vol.2, ed. R.T. Yeh, Prentice-Hall, 1977, pp.44-79.
- [6] J.B. Goodenough, "The Ada Compiler Validation Capability," Proc. ACM SIGPLAN Symp. on the Ada Programming Language, Boston, Dec 1980.
- [7] R.G. Hamlet, "Critique of Reliability Theory," Proc IEEE Workshop on Software Testing and Test Documentation, Fort Lauderdale, Dec 1978, pp.57-69.
- [8] W.E. Howden, "Methodology for the Generation of Program Test Data," IEEE Trans. on Computers, Vol.C-24, May 1975, pp. 554-559.
- [9] W.E. Howden, "Theoretical and Empirical Studies on Program Testing," IEEE Trans. Software Eng, SE-4, July 1978, pp.293-298.
- [10] W.E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Trans. Software Eng., Vol.SE-2, Sept 1976, pp.208-215.
- [11] W.E. Howden, "Algebraic Program Testing," Acta Informatica, Vol.10, 1978, pp.53-66.
- [12] W.E. Howden, "Functional Program Testing," IEEE Trans. Software Eng., Vol.SE-6, March 1980, pp.162-169.
- [13] W.E. Howden, "Completeness Criteria for Testing Elementary Program Functions," U. Victoria Dept. of Mathematics, DM-212-IR, May 1980.
- [14] K.C. Tai, "On Program Testing Criteria," Proc CompSAC 79, Chicago, Nov 1979, pp.695-701.

[15] T.A. Thayer, M. Lipow, and E.C. Nelson, Software Reliability, A Study of Large Project Reality, North Holland Publishing Co., New York, 1978.

[16] E.J. Weyuker, Program Schemas with Semantic Restrictions, Dept of Computer Science Technical Rpt No. 60, Rutgers University, New Brunswick, New Jersey, May 1977, 168 pages. (Ph.D. Thesis)

[17] E.J. Weyuker, "Translatability and Decidability Questions for Restricted Classes of Program Schemas," SIAM J. Comput., Vol.8, 1979.

[18] E.J. Weyuker, "The Applicability of Program Schema Results to Programs," Int. J. Comput. Inf. Sci., Vol.8, No.5, Nov 1979.

[19] E.J. Weyuker, "The Oracle Assumption of Program Testing," Proc. of the 13th Hawaii International Conf on System Sciences, Hawaii, Jan 1980.

[20] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Trans. Software Eng., Vol.SE-6, May 1980, pp.236-246.

[21] L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Trans. Software Eng., Vol. SE-6, May 1980, pp.247-257.

This book may be kept

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

GAYLORD 142			PRINTED IN U.S.A.

NYU c.2
Comp. Sci. Dept.
TR-027
Weyuker
An error-based testing
strategy.

NYU c.2	
Comp. Sci. Dept.	
TR-027	
Weyuker	
AUTHOR	
An error-based testing	
TITLE	
strategy.	
DATE DUE	BORROWER'S NAME

**N.Y.U. Courant Institute of
Mathematical Sciences**
251 Mercer St.
New York, N. Y. 10012

